ELEX 7660: Digital System Design

Project Report

Automatic Garage Door Opener

Armin Laghaee, Kevin Shu

April 15, 2017

Abstract

This report discusses the digital design of an automatic garage door system using RF technology. The report will also explain both the hardware and software design processes used to implement the model.

Table of Contents

1 Introduction

This project was prepared for Dr. Eduardo (Ed) Casas for ELEX 7660 - Digital System Design [1]. In this project, we modeled an auto garage door opener using the RF transmitter and receiver technology. We used the DE0-Nano FPGA and System Verilog as the programming language. Our goal was to simulate an everyday garage door.

2 Background

The garage door opener is a motorized device which will close or open the garage door of your house. Most garage door openers are controlled by a remote controller. The remoter controller sends a signal wirelessly to the garage door opener. The garage door revises the signal and does an action. Specifically, if the garage door is closed and the home owner wants to get out or get in the house, he/she will press the button on the controller so the garage opener will activate the motor to open the door. The same action is done to close the garage doors. In addition, the garage door will pause if the owner presses the button while closing or opening.

Figure 1: A modern garage door system

3 Procedure

Both hardware and software designs were used in this project. Circuits for the remote controller and the garage system were designed and implemented. Specifically, components such as the HT12E, the encoder, were used to transmit data. On the receiver side, the HT12D was used to decode the signal and giving it to the FPGA. The DE0-Nano was the FPGA which we used for this project. Later, the FPGA sends the appropriate 1's and 0's parallel to the stepping motor, so the motor can rotate clockwise or counter-clockwise. We first used breadboards to check if the circuit was working properly. After finding some mistakes and troubleshooting, we soldered the components of the controller and the garage system on two different circuit boards.

Regarding the software, we programmed the FPGA with System Verilog. Different modules were created to accomplish different parts of the objective. To illustrate, we used one module to control the motion of the stepping motor, and one to track the time and create delay, and another which indicates when to turn on or off the motor. After many research and studying, we concluded that the best way to meet our objectives is to create and use state machines. After drawing flow charts and diagram, we wrote our code on ModelSim. Making sure there are no syntax errors, we used Quartus Prime 16.1 to compile and synthesize. In addition, a .qsf file was used to set the pins of the FPGA. We were able to resolve many of our logic errors by creating test-bench for most of our modules.

4 Circuit Design

As mentioned earlier we designed two circuits. One is for the controller which will send a signal when the push button on the controller is pressed, and one is the garage system which will receive the signal and runs the stepper motor.

4.1 Controller Circuit

The controller basically sees the position of the push button switch and can send a signal with clock trough the transmitter. The controller circuit consists of a regulator, H12E, and RF Link transmitter. Below we will describe each of them in detail.

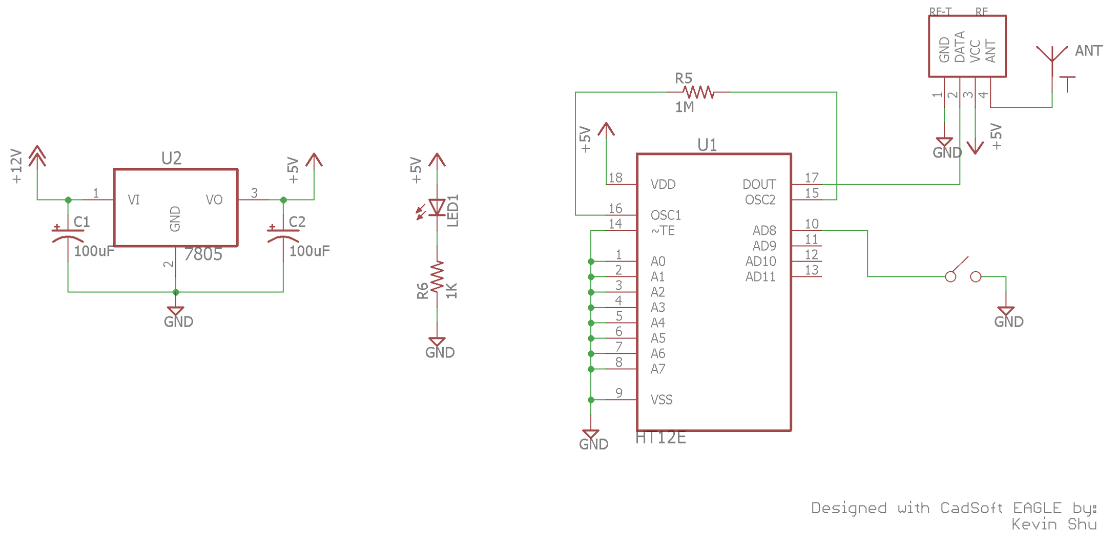


Figure 2: Controller Circuit Diagram

4.1.1 Voltage Regulator

We used 9V batteries to power our remote controller, but we need a constant 5V for the HT12E IC. For this reason, we used the 7805 voltage regulator to output a constant 5V. The figure below shows how the regulator was wired. We used capacitors for better performance and add safety to the regulator.

4.1.2 LED Power indicator

In our controller, we simply used a LED to indicate if the controller has power. If the LED is not on, it indicates that there is no voltage in our controller, or the amount of the voltage is not adequate.

4.1.3 Encoder

The HT12E was used to encode the information and transmit the addresses/data programmed with the header bit via an RF by receipt of a trigger signal. In this design the HT12E is always activated; thus, $\overline{TE}$ is connected to the ground. The HT12E acts like a VCO. This means applying a specific voltage and choosing a resistor value, we can obtain the desired frequency which we want to send to the receiver. In our design, we used a 5V and 1M ohm resistor to create a 3kHz frequency clock. For more information on how the frequency was chosen refer to Appendix A.1 or the data sheet.
We connected an active high push button to one of the address/data input pins, specifically AD8. So when the PB switch is pressed it will output a high from pin 17 which will be connected to the transmitter. The address pins were all left grounded.

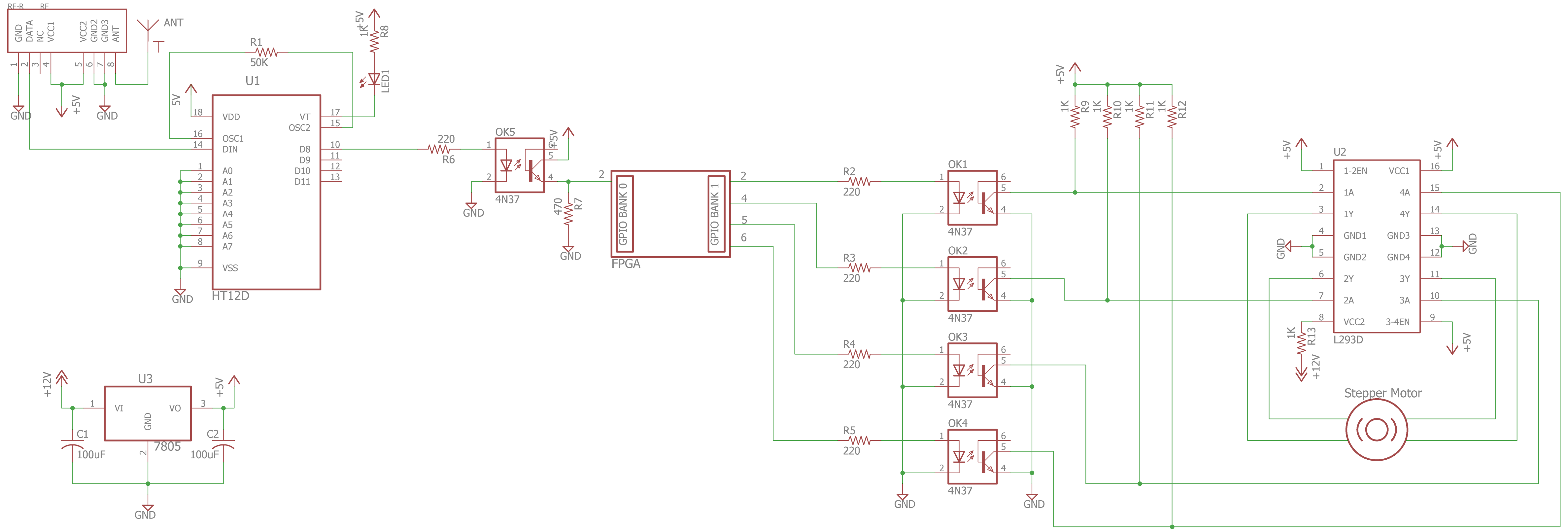
4.1.4 RF-Transmitter

The RF transmitter (4800bps) is used to create a very simple wireless data link. This creates a simplex channel. In other words, the transmitter can only send the signal, not receiving a signal.

4.2 Garage System Circuit

The garage system receives the signal and decodes it. Using the FPGA, the garage system will run the stepping motor. The figure below is the overall circuit diagram of the garage system.

GARAGE SYSTEM



RF-R RF
GND DATA NC VCC1 VCC2 GND2 GND3 ANT
ANT
R1 50K
U1
+5V 1K R8
LED1
VDD OSC1 DIN VT OSC2 D8 D9 D10 D11
A0 A1 A2 A3 A4 A5 A6 A7 VSS
HT12D
220 R6
OK5 4N37
470 R7
GPIO BANK 0 GPIO BANK 1
FPGA
R2 220 R3 220 R4 220 R5 220
OK1 OK2 OK3 OK4 4N37
1K R9 1K R10 1K R11 1K R12
U2 L293D
1-2EN 1A 1Y GND1 GND2 2Y 2A VCC2 VCC1 4A 4Y GND3 GND4 3Y 3A 3-4EN
1K R13 +12V
Stepper Motor
U3 VI VO GND 7805
C1 100uF C2 100uF
+12V +5V GND

Designed with CadSoft EAGLE by:
Kevin Shu

4.2.1 RF-Receiver

In this project, the RF receiver (4800bps) was used to receive the signal from the controller. It is important to note that the receiver also requires a 5V supply voltage.

4.2.2 Decoder

The H12D was used to decode the received serial addresses and data from the encoder. Just like the encoder, we used 5V and a resistor value, 50KΩ in this case, to generate the desired frequency. We want our decoder frequency to be set to 150 kHz because it is recommended that our decoder oscillator is **50 times** the encoder frequency.

If the decoder gets a signal, D0 (pin 10) will be high, and a LED will go on as it can be seen from the circuit diagram above.

4.2.3 Optocoupler

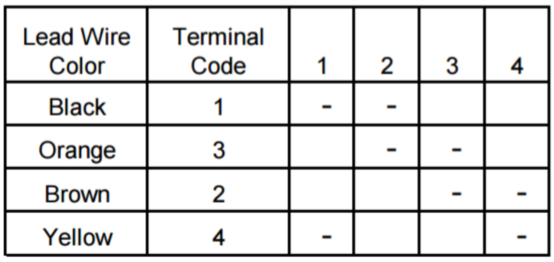
We used the 4N35 optocouplers to isolate the circuits from both sides of the FPGA. This is to make sure there is no high current that will damage the FPGA. The optocoupler consists of an IR-diode and a phototransistor. The phototransistor will turn on if it sees the IR-diode lights up which will activate its circuit.

4.2.4 Half-H Driver

The L293D was used to provide bi-directional current to the stepping motor. In other words, it will cause the motor to run clockwise or counter-clockwise. The half-H driver requires a 5V and a 12V to drive the stepper motor. In general, when an input is high, the associated output is high, and when the input is low the output is low [2].

4.2.5 Stepper Motor

In this project, we used an ST-PM35-15-11C stepper motor. The motor is a bipolar stepper motor, which consists of 4 wires. To make the motor to rotate clockwise we have to follow the pole excitation order in Figure 3.



Lead Wire Color	Terminal Code	1	2	3	4
Black	1	-	-		
Orange	3		-	-	
Brown	2			-	-
Yellow	4	-			-

Figure 3: Pole excitation required to rotate stepper motor clockwise

Note that in Figure 3, a "–" means a logic 1. Also, if you want to rotate the motor counter-clockwise, you have to reverse the order, which means you have to do terminal code 4,3,2,1 in order.

5 Modular Design

As mentioned earlier we used System Verilog to program the FPGA. We created the following modules for the automatic garage door. Their codes can be found in Appendix B of the report.

5.1 *Project1* Module

5.1.1 Overview

This is the top-level entity, main module, that has the following inputs:

`A`	input pin on the FPGA to receive the signal from the controller (the transmitter)
`KEY[0]`	push button on the FPGA used to reset the program
`KEY[1]`	If we are not using the controller (not in RF mode), we can use the push button on the FPGA
`CLOCK_50`	Using the 50 MHz of the FPGA as our initial clock

In addition, we are using the following outputs:

`out`	is a 4-bit output to run the stepping motor
`clk2kHz`	is a 2kHz clock created by the PLL
`clk48Hz`	is a 48Hz (60RPM) clock that the stepper is running at
`PB_sync`	because we are using push button on the controller, we need a synchronize the PB to our 48Hz clock

Figure 4 shows the overall block diagram of the top-level module. We will describe some of the input and output logics in more detail below.

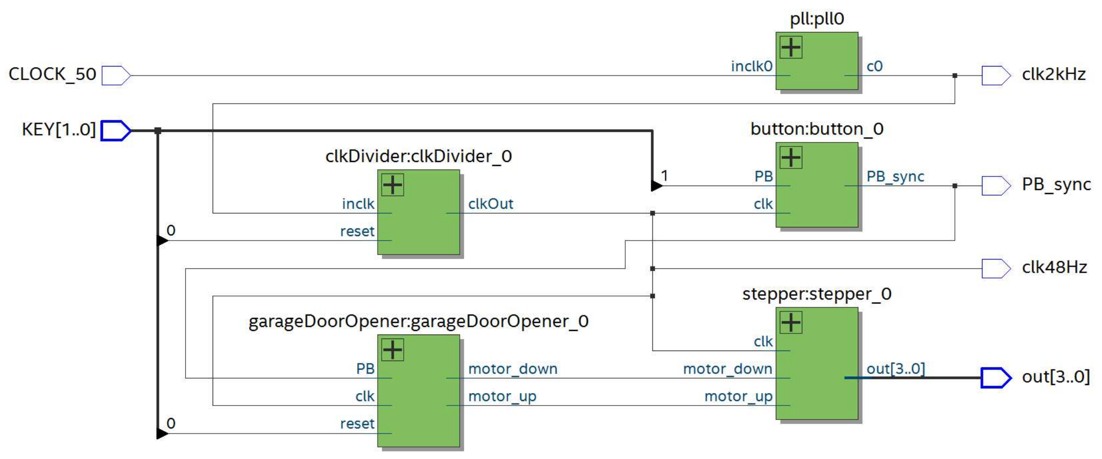


Figure 4: Block Symbol of *Project1*

5.1.2 Clock Division

A 50MHz clock is too fast for the stepper motor to run. In other words, the motor cannot get 1's and 0's that fast. The stepper motor needs 48 steps or pulses to complete 1 revolution. In other words, 48 pulses applied in 1 sec, the motor will complete 1 revolution, this is equivalent to 60 RPM. Please refer to Section 5.2.3 on how we got 48 steps.

So in our *clkDivider* module, we used the Altera Phase Lock Loop (PLL) to convert the 50MHz down to a 2kHz clock because the PLL has a limited divider range. Therefore, we used a counter to further divide the clock to 48Hz.

This means it will take 20.83ms to do a step and therefore requires to divide the 2kHz clock by 41.66. The number of bits required for `count` would be 6 bits (shown in equation 3). We need to make sure to have the duty cycle of the clock at 50%. Thus, the `period` was divided in half where half of it was set high and other half was set low.

$$delay = \frac{1}{48Hz} = 20.83ms \tag{1}$$

$$frequency\ divider = 2000Hz \times 20.83ms = 41.66 \tag{2}$$

$$number\ of\ bits = \frac{\log(41.66)}{\log(2)} = 5.38 = 6\ bits \tag{3}$$

5.1.3 Instantiation

Our *project1* is our top-level module where we instantiate all of the other modules here (*stepper*, *garageDoorOpener*). The input, output, and other variables defined in the *project1* module, such as `motor_up` and `motor_down`, are assigned to the inputs and outputs of different modules. Below is an example of how we instantiate a module.

```
button button_0(
.clk(clk48Hz),
.PB(KEY[1]),
.PB_sync(PB_sync)
);
```

In the example, we put the 48Hz clock we created and the `KEY[1]`, input of *project1*, as the *button* module's input, and `PB_sync`, the output of *project1* module, as *button* module's output.

5.2 *garageDoorOpener* Module

5.2.1 Module Behaviour

Based on the clock and the input signal received from the push button, this module indicates which state our stepping motor is at. This module is basically a state machine. The logic of this state machine is shown in Figure 5.

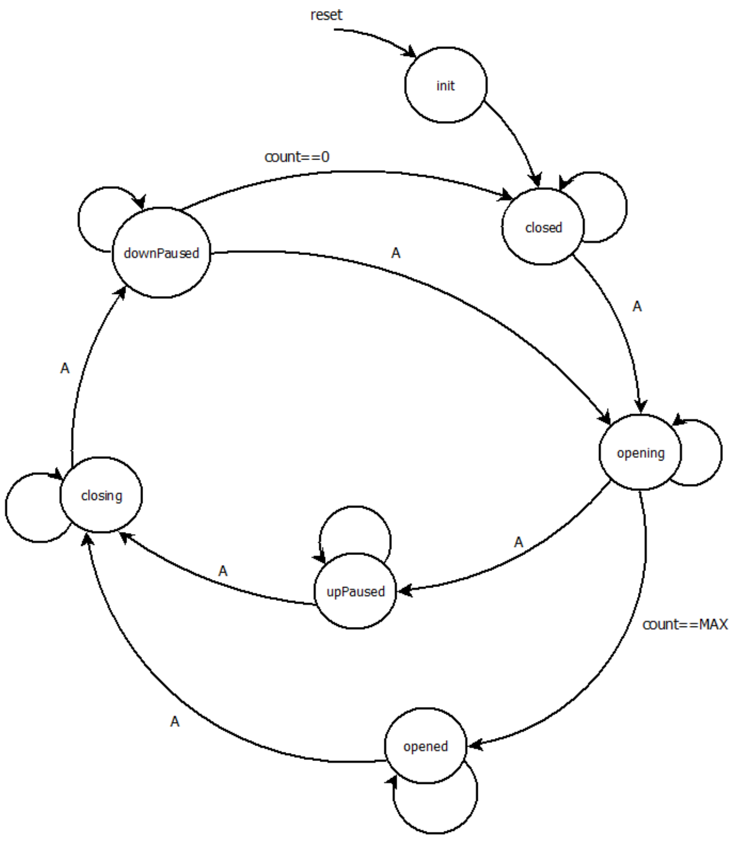


Figure 5: State Diagram of the *garageDoorOpener* module

5.2.2 Describing the State Machine

When resetting, the *garageDoorOpener* module will go to the initialization (`init`) state, which will make sure that motor is off. We programmed it so after `init` state it will go to the `closed` state.

When the garage door is `opened` or `closed`, it will indicate it by using a flag (`atTop`, `atBottom`). In addition, it tells the stepper module to output `0` so the motor will not move. Using an if-statement (if the push button is pressed) it will go to the next state of the state machine.

In the `closing` and `opening` state, the FPGA will turn the motor clockwise or counter-clockwise. In our setup, when the door is closing, the motor turns counter-clockwise, and if the motor is opening, the motor turns clockwise. During the time that the door is closing or opening, if you press the push button the door will stop moving. If you press the push button again, the door will move in the opposite direction.
In general, in each state, we indicated the flags and used if-statements to transition to the next state.

5.2.3 Range of the Counter

It is also important to note that we wanted our stepping motor to rotate a complete cycle either clockwise or counter-clockwise. To do so, we need to know each step moves by how many degrees. The stepping motor we used is rated to have a stride angle of 7.5 degrees for each step and therefore require 48 steps to complete a revolution.

$$maxGarageStep = \frac{360^\circ}{7.5^\circ} = 48 \ steps \ per \ revolution \tag{4}$$

Our counter counts down from `maxGarageStep` to zero in closing state and counts up from zero to `maxGarageStep` in the opening state. From trial and error, we determined it takes 400 steps to completely open our garage door.

5.3 *Stepper* Module

5.3.1 Module Behaviour

This module consists of a simple if-else statement. Basically, this module checks if `motor_up` or `motor_down` flag is set. If `motor_down` is set the motor will move counter-clockwise, as a result, we will see that the door is closing. If the `motor_up` is set the motor will move clockwise, and so the door is opening.

5.3.2 Motor Direction

To make the motor to move, a sequence of a 4-bit value is applied to the motor, since we are using a 4-wire bipolar stepping motor. The sequence of ones and zeros applied to the stepping motor was mentioned in Section 4.2.5 of the report.

To apply this logic in System Verilog, a 4-bit variable called `out` was defined which gets each row of the `up` and sends it out of the FPGA to turn the motor. Each row consists of a 4-bit value. As it can be seen from the Figure 6 below, if we go from `up[0]` to `up[3]` the motor will go clockwise, and if we go from `up[3]` to `up[0]` the motor will go counter-clockwise.

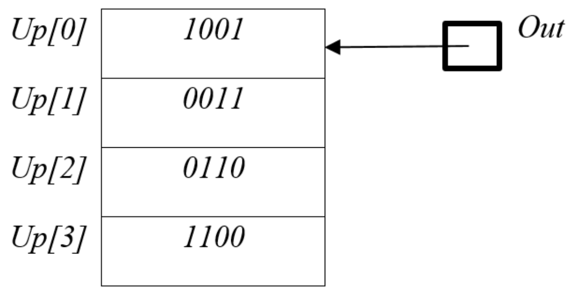


Figure 6: Sequence of bits to make the motor to move

5.4 *Button* Module

5.4.1 Module Behaviour

This module ensures that the push button is in synchronous with the clock at 48Hz. 48Hz is fast and since our states change on every positive edge of the clock, we have to make sure that the moment you push the button happens on the edge and stays high till the end of the clock cycle. Or else, the state machine will go through multiple states as you hold onto the push button.

As we used the pushbutton on the FPGA (active low), we invert the signal as shown below in Appendix B.4.

5.5 Garage Model

Figure 7 shows our model of the garage system where we used thin wooden panels to construct the walls and roof. The stepper motor was positioned above and centered so that the garage door can move up with ease. There was also plastic guide rails to help guide the door up. The door was cut from cardboard as it was light. We used tooth floss as the string.

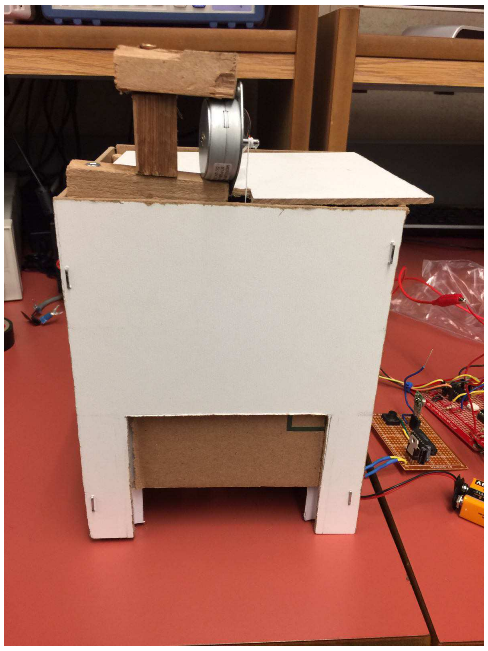


Figure 7: Our Garage System Model

6 Conclusion

This report has shown the hardware and software design of the automatic garage door opener using RF technology. However, more future work can be added to the project such as a timer where it counts down the time in case the homeowner forgets to close the door. Also, sensors can be incorporated into the system where can detect for any objects in the way of the garage door. A LED display can be used to show messages and information.

References

[1] E. Casas, *Elex 7660 course notes*. 2017.

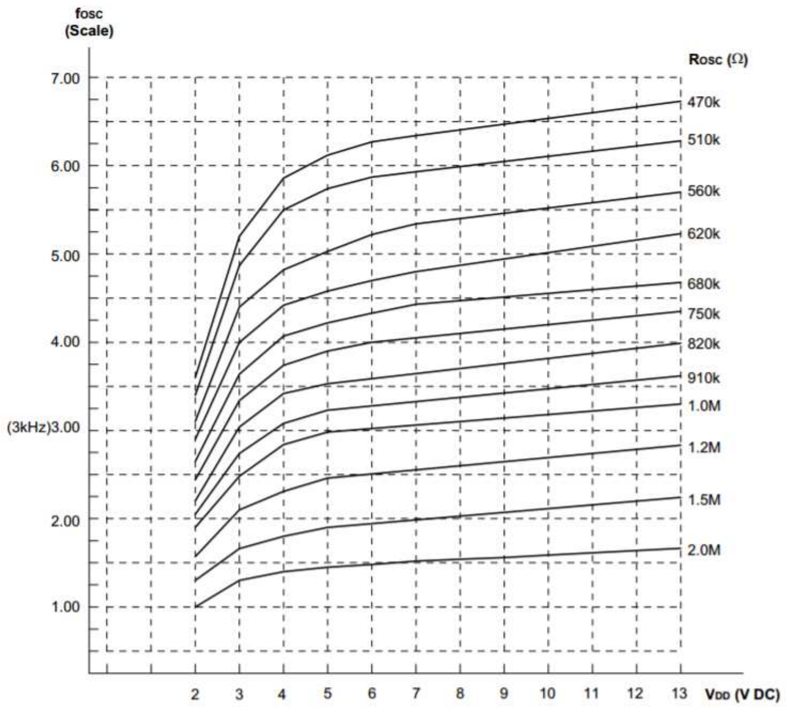
[2] T. Instruments. (). L293x quadruple half-h drivers, [Online]. Available: `http://www.ti.com/lit/ds/symlink/l293.pdf`.

[3] Holtek. (). Ht12a/ht12e series of encoders, [Online]. Available: `https://www.engineersgarage.com/sites/default/files/HT12E_0.PDF`.

[4] ——, (). Ht12d/ht12f series of decoders, [Online]. Available: `http://www.farnell.com/datasheets/1525377.pdf`.

7 Appendix A: Frequency Datasheets

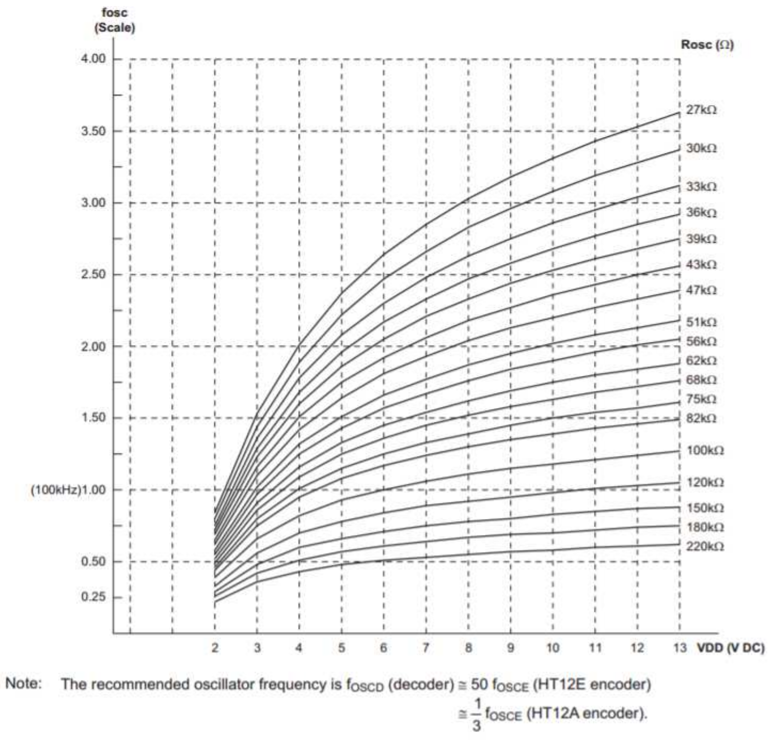
7.1 A.1: HT12E Oscillator Frequency



The recommended oscillator frequency is f_{OSCD} (decoder) $\cong 50\ f_{OSCE}$ (HT12E encoder)
$\cong \frac{1}{3} f_{OSCE}$ (HT12A encoder)

Figure 8: Oscillator frequency vs supply voltage of HT12E [3].

7.2 A.2: HT12D Oscillator Frequency



Note: The recommended oscillator frequency is f_{OSCD} (decoder) $\cong$ 50 f_{OSCE} (HT12E encoder) $\cong \frac{1}{3} f_{OSCE}$ (HT12A encoder).

Figure 9: Oscillator frequency vs supply voltage of HT12D [4].

8 Appendix B: Code

8.1 B.1: *project1* Module

```
// project1.sv - Automatic Garage Door Opener top-level module
// Armin Laghaee & Kevin Shu
// Date: 2017-04-14

module project1 ( output logic [3:0] out,  // motor output
          output logic clk48Hz,
          output logic clk2kHz,
          output logic PB_sync,
          input logic A,  // I/P for from RF RX
          input logic [1:0] KEY,
          input logic CLOCK_50
          );

   logic motor_up, motor_down;

   // instantiate your modules here...

  pll pll0(
    .inclk0(CLOCK_50),
    .c0(clk2kHz)
    );

  clkDivider clkDivider_0(
    .inclk(clk2kHz),
    .reset(KEY[0]),
    .clkOut(clk48Hz)
    );

  button button_0(
    .clk(clk48Hz),
    .PB(KEY[1]),
    .PB_sync(PB_sync)
    );

   garageDoorOpener garageDoorOpener_0(
    .clk(clk48Hz),
    .PB(PB_sync),
    .reset(KEY[0]),
    .motor_up(motor_up),
    .motor_down(motor_down)
    );

   stepper stepper_0(
    .clk(clk48Hz),
    .motor_up(motor_up),
    .motor_down(motor_down),
    .out(out)
    );
endmodule


module clkDivider (input logic inclk, reset,
            output logic clkOut);

  parameter period = 48; //48Hz = 60 RPM creating 20.83ms clock
  parameter N = 5; // bits - 1

  parameter halfPeriod = period / 2;
  logic [N:0] count;

  always_ff @(posedge inclk)
  begin
    if (!reset) begin
      count <= 0;
      clkOut <= 0;
     end
     else begin
      if (count > halfPeriod - 1) begin
        count <= 0;
        clkOut <= ~clkOut;
      end
      else count <= count + 1;
    end
  end
endmodule

module pll (inclk0, c0);

    input inclk0;
    output c0;

    wire [0:0] sub_wire2 = 1'h0;
    wire [4:0] sub_wire3;
    wire  sub_wire0 = inclk0;
    wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
    wire [0:0] sub_wire4 = sub_wire3[0:0];
    wire  c0 = sub_wire4;

    altpll altpll_component ( .inclk (sub_wire1), .clk
     (sub_wire3), .activeclock (), .areset (1'b0), .clkbad
     (), .clkena ({6{1'b1}}), .clkloss (), .clkswitch
     (1'b0), .configupdate (1'b0), .enable0 (), .enable1 (),
     .extclk (), .extclkena ({4{1'b1}}), .fbin (1'b1),
     .fbmimicbidir (), .fbout (), .fref (), .icdrclk (),
     .locked (), .pfdena (1'b1), .phasecounterselect
     ({4{1'b1}}), .phasedone (), .phasestep (1'b1),
     .phaseupdown (1'b1), .pllena (1'b1), .scanaclr (1'b0),
     .scanclk (1'b0), .scanclkena (1'b1), .scandata (1'b0),
     .scandataout (), .scandone (), .scanread (1'b0),
     .scanwrite (1'b0), .sclkout0 (), .sclkout1 (),
     .vcooverrange (), .vcounderrange ());

    defparam
          altpll_component.bandwidth_type = "AUTO",
          altpll_component.clk0_divide_by = 25000,
          altpll_component.clk0_duty_cycle = 50,
          altpll_component.clk0_multiply_by = 1,
          altpll_component.clk0_phase_shift = "0",
          altpll_component.compensate_clock = "CLK0",
          altpll_component.inclk0_input_frequency = 20000,
          altpll_component.intended_device_family = "Cyclone IV E",
          altpll_component.lpm_hint = "CBX_MODULE_PREFIX=lab1clk",
          altpll_component.lpm_type = "altpll",
          altpll_component.operation_mode = "NORMAL",
          altpll_component.pll_type = "AUTO",
          altpll_component.port_activeclock = "PORT_UNUSED",
          altpll_component.port_areset = "PORT_UNUSED",
          altpll_component.port_clkbad0 = "PORT_UNUSED",
          altpll_component.port_clkbad1 = "PORT_UNUSED",
          altpll_component.port_clkloss = "PORT_UNUSED",
          altpll_component.port_clkswitch = "PORT_UNUSED",
          altpll_component.port_configupdate = "PORT_UNUSED",
          altpll_component.port_fbin = "PORT_UNUSED",
          altpll_component.port_inclk0 = "PORT_USED",
          altpll_component.port_inclk1 = "PORT_UNUSED",
          altpll_component.port_locked = "PORT_UNUSED",
          altpll_component.port_pfdena = "PORT_UNUSED",
          altpll_component.port_phasecounterselect = "PORT_UNUSED",
          altpll_component.port_phasedone = "PORT_UNUSED",
          altpll_component.port_phasestep = "PORT_UNUSED",
          altpll_component.port_phaseupdown = "PORT_UNUSED",
          altpll_component.port_pllena = "PORT_UNUSED",
          altpll_component.port_scanaclr = "PORT_UNUSED",
          altpll_component.port_scanclk = "PORT_UNUSED",
          altpll_component.port_scanclkena = "PORT_UNUSED",
          altpll_component.port_scandata = "PORT_UNUSED",
          altpll_component.port_scandataout = "PORT_UNUSED",
          altpll_component.port_scandone = "PORT_UNUSED",
          altpll_component.port_scanread = "PORT_UNUSED",
          altpll_component.port_scanwrite = "PORT_UNUSED",
          altpll_component.port_clk0 = "PORT_USED",
          altpll_component.port_clk1 = "PORT_UNUSED",
          altpll_component.port_clk2 = "PORT_UNUSED",
          altpll_component.port_clk3 = "PORT_UNUSED",
          altpll_component.port_clk4 = "PORT_UNUSED",
          altpll_component.port_clk5 = "PORT_UNUSED",
          altpll_component.port_clkena0 = "PORT_UNUSED",
          altpll_component.port_clkena1 = "PORT_UNUSED",
          altpll_component.port_clkena2 = "PORT_UNUSED",
          altpll_component.port_clkena3 = "PORT_UNUSED",
          altpll_component.port_clkena4 = "PORT_UNUSED",
          altpll_component.port_clkena5 = "PORT_UNUSED",
          altpll_component.port_extclk0 = "PORT_UNUSED",
          altpll_component.port_extclk1 = "PORT_UNUSED",
          altpll_component.port_extclk2 = "PORT_UNUSED",
          altpll_component.port_extclk3 = "PORT_UNUSED",
          altpll_component.width_clock = 5;
  endmodule
```

8.2 B.2: *garageDoorOpener* Module

```
// garageDoorOpener.sv - This module is a state machine for the garage door opener.
    It cycles through the states when PB is pushed
// Armin Laghaee & Kevin Shu
// Date: 2017-04-14


module garageDoorOpener (input logic clk, PB, reset,
              output logic motor_up, motor_down, paused);

  parameter maxGarageStep = 400; // 48 steps per rev., motor has 7.5 stride angle

  logic atTop, atBottom = 0;
  enum logic [6:0] {init, opened, closed, opening, closing, upPaused, downPaused}
    state, state_next;
  logic [31:0] count, count_next = 0;

  // controller state
  always_ff @(posedge clk or negedge reset) begin
    if (!reset)
      state <= init;
    else begin
      state <= state_next;
      count <= count_next;
    end
  end

  // datapath logic
  always_comb begin
    state_next = state;
    count_next = count;

    case (state)
      init: begin
        atTop = 0; atBottom = 0;
        motor_up = 0; motor_down = 0;
        state_next = closed;
      end

      opened: begin
        atTop = 1; atBottom = 0;
        motor_up = 0; motor_down = 0;
        count_next = maxGarageStep;
        if (PB)
          state_next = closing;
        else
          state_next = opened;
      end

      closing: begin
        atTop = 0; atBottom = 0;
        motor_up = 0; motor_down = 1;
        if (PB)
          state_next = downPaused;
        else if (count == 0)
          state_next = closed;
        else begin
          count_next = count - 1;
        end
      end

      downPaused: begin
        atTop = 0; atBottom = 0;
        motor_up = 0; motor_down = 0;
        if (PB)
          state_next = opening;
        else
          state_next = downPaused;
      end

      closed: begin
        atTop = 0; atBottom = 1;
        motor_up = 0; motor_down = 0;
        count_next = 0;
        if (PB)
          state_next = opening;
        else
          state_next = closed;
      end

      opening: begin
        atTop = 0; atBottom = 0;
        motor_up = 1; motor_down = 0;
        if (PB)
          state_next = upPaused;
        else if (count == maxGarageStep)
          state_next = opened;
        else begin
          count_next = count + 1;
        end
      end

      upPaused: begin
        atTop = 0; atBottom = 0;
        motor_up = 0; motor_down = 0;
        if (PB)
          state_next = closing;
        else
          state_next = upPaused;
      end
    endcase
  end
endmodule
```

8.3 B.3: *stepper* Module

```
// stepper.sv - This module allows the stepper motor to rotate clockwise or counter-
    clockwise according to the pole excitation order
// Armin Laghaee & Kevin Shu
// Date: 2017-04-14

module stepper (input logic clk, motor_up, motor_down,
          output logic [3:0] out);

  logic [1:0] step, step_next;
  logic [3:0] out_next;
  logic [3:0] up[4] = '{ 4'b1001, 4'b0011, 4'b0110, 4'b1100};

  always_ff @(posedge clk) begin
    step <= step_next;
    out <= out_next;
  end

  always_comb begin
    out_next = out;
    step_next = step;

    if (motor_up) begin
      if (step < 4) begin
        out_next = up[step];
        step_next = step + 1;
      end
      else
        step_next = 0;
    end
    else if (motor_down) begin
      if (step < 4) begin
        out_next = up[step];
        step_next = step - 1;
      end
      else
        step_next = 0;
    end
  end
endmodule
```

8.4 B.4: *button* Module

```
// button.sv - This module syncs the button to clock and gets the rising edge
// Armin Laghaee & Kevin Shu
// Date: 2017-04-14

module button (input logic clk, PB,
          output logic PB_sync);

  logic a, b, c;

  always_ff @(posedge clk) begin
    a <= ~PB;
    b <= a;
    c <= b;
  end

  always_comb begin
    PB_sync = (~c) & b; // get rising edge
  end

endmodule
```